

# **EXHIBIT E**

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

1  /*
2   * Copyright (C) 2008 The Android Open Source Project
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
7   *
8   *     http://www.apache.org/licenses/LICENSE-2.0
9   *
10  * Unless required by applicable law or agreed to in writing, software
11  * distributed under the License is distributed on an "AS IS" BASIS,
12  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
13  * implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  */
17 package java.util;
18
19 /**
20  * This is a near duplicate of {@link TimSort}, modified for use with
21  * arrays of objects that implement {@link Comparable}, instead of using
22  * explicit comparators.
23  *
24  * <p>If you are using an optimizing VM, you may find that
25  * ComparableTimSort
26  * offers no performance benefit over TimSort in conjunction with a
27  * comparator that simply returns {@code
28  * ((Comparable)first).compareTo(Second)}.
29  * If this is the case, you are better off deleting ComparableTimSort to
30  * eliminate the code duplication. (See Arrays.java for details.)
31  */
32 class ComparableTimSort {
33     /**
34      * This is the minimum sized sequence that will be merged. Shorter
35      * sequences will be lengthened by calling binarySort. If the
36      * entire
37      * array is less than this length, no merges will be performed.
38      *
39      * This constant should be a power of two. It was 64 in Tim Peter's
40      * C
41      * implementation, but 32 was empirically determined to work better
42      * in
43      * this implementation. In the unlikely event that you set this
44      * constant
45      * to be a number that's not a power of two, you'll need to change
46      * the
47      * {@link #minRunLength} computation.
48      *
49      * If you decrease this constant, you must change the stackLen
50      * computation in the TimSort constructor, or you risk an
51      * ArrayOutOfBoundsException exception. See listsort.txt for a discussion
52      * of the minimum stack length required as a function of the length
53      * of the array being sorted and the minimum merge sequence length.
54      */
55     private static final int MIN_MERGE = 32;
56
57     /**
58      * The array being sorted.
59      */

```

UNITED STATES DISTRICT COURT  
NORTHERN DISTRICT OF CALIFORNIA  
**TRIAL EXHIBIT 45.2**  
CASE NO. 10-03561 WHA  
DATE ENTERED\_\_\_\_\_  
BY\_\_\_\_\_  
DEPUTY CLERK

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

53     private final Object[] a;
54
55     /**
56      * When we get into galloping mode, we stay there until both runs
57      * win less
58      * often than MIN_GALLOP consecutive times.
59      */
60     private static final int MIN_GALLOP = 7;
61
62     /**
63      * This controls when we get *into* galloping mode. It is
64      * initialized
65      * to MIN_GALLOP. The mergeLo and mergeHi methods nudge it higher
66      * for
67      * random data, and lower for highly structured data.
68      */
69     private int minGallop = MIN_GALLOP;
70
71     /**
72      * Maximum initial size of tmp array, which is used for merging.
73      * The array
74      * can grow to accommodate demand.
75      *
76      * Unlike Tim's original C version, we do not allocate this much
77      * storage
78      * when sorting smaller arrays. This change was required for
79      * performance.
80      */
81     private static final int INITIAL_TMP_STORAGE_LENGTH = 256;
82
83     /**
84      * Temp storage for merges.
85      */
86     private Object[] tmp;
87
88     /**
89      * A stack of pending runs yet to be merged. Run i starts at
90      * address base[i] and extends for len[i] elements. It's always
91      * true (so long as the indices are in bounds) that:
92      *
93      *     runBase[i] + runLen[i] == runBase[i + 1]
94      *
95      * so we could cut the storage for this, but it's a minor amount,
96      * and keeping all the info explicit simplifies the code.
97      */
98     private int stackSize = 0; // Number of pending runs on stack
99     private final int[] runBase;
100    private final int[] runLen;
101
102    /**
103     * Asserts have been placed in if-statements for performance. To
104     * enable them,
105     * set this field to true and enable them in VM with a command line
106     * flag.
107     * If you modify this class, please do test the asserts!
108     */
109     private static final boolean DEBUG = false;
110
111    /**
112     * Creates a TimSort instance to maintain the state of an ongoing

```

G:\eclair21 - GOOGLE-00-00000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

```
105
106     *
107     * @param a the array to be sorted
108     */
109     private ComparableTimSort(Object[] a) {
110         this.a = a;
111
112         // Allocate temp storage (which may be increased later if
113         // necessary)
114         int len = a.length;
115         @SuppressWarnings({ "unchecked", "UnnecessaryLocalVariable" })
116         Object[] newArray = new Object[len < 2 * INITIAL_TMP_STORAGE_LENGTH ?
117                                         len >>> 1 :
118                                         INITIAL_TMP_STORAGE_LENGTH];
119         tmp = newArray;
120
121         /*
122          * Allocate runs-to-be-merged stack (which cannot be expanded).
123          * The
124          * stack length requirements are described in listsort.txt.  The
125          * C
126          * version always uses the same stack length (85), but this was
127          * measured to be too expensive when sorting "mid-sized" arrays
128          * (e.g.,
129          * 100 elements) in Java.  Therefore, we use smaller (but
130          * sufficiently
131          * large) stack lengths for smaller arrays.  The "magic numbers"
132          * in the
133          * computation below must be changed if MIN_MERGE is decreased.
134          * See
135          * the MIN_MERGE declaration above for more information.
136          */
137         int stackLen = (len < 120 ? 5 :
138                         len < 1542 ? 10 :
139                         len < 119151 ? 19 : 40);
140         runBase = new int[stackLen];
141         runLen = new int[stackLen];
142     }
143
144     /*
145      * The next two methods (which are package private and static)
146      * constitute
147      * the entire API of this class.  Each of these methods obeys the
148      * contract
149      * of the public method with the same signature in java.util.Arrays.
150      */
151
152     static void sort(Object[] a) {
153         sort(a, 0, a.length);
154     }
155
156     static void sort(Object[] a, int lo, int hi) {
157         rangeCheck(a.length, lo, hi);
158         int nRemaining = hi - lo;
159         if (nRemaining < 2)
160             return; // Arrays of size 0 and 1 are always sorted
161
162         // If array is small, do a "mini-TimSort" with no merges
163         if (nRemaining < MIN_MERGE) {
```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

153         int initRunLen = countRunAndMakeAscending(a, lo, hi);
154         binarySort(a, lo, hi, lo + initRunLen);
155         return;
156     }
157
158     /**
159      * March over the array once, left to right, finding natural
160      * runs,
161      * extending short natural runs to minRun elements, and merging
162      * runs
163      * to maintain stack invariant.
164      */
165     ComparableTimSort ts = new ComparableTimSort(a);
166     int minRun = minRunLength(nRemaining);
167     do {
168         // Identify next run
169         int runLen = countRunAndMakeAscending(a, lo, hi);
170
171         // If run is short, extend to min(minRun, nRemaining)
172         if (runLen < minRun) {
173             int force = nRemaining <= minRun ? nRemaining : minRun;
174             binarySort(a, lo, lo + force, lo + runLen);
175             runLen = force;
176         }
177
178         // Push run onto pending-run stack, and maybe merge
179         ts.pushRun(lo, runLen);
180         ts.mergeCollapse();
181
182         // Advance to find next run
183         lo += runLen;
184         nRemaining -= runLen;
185     } while (nRemaining != 0);
186
187     // Merge all remaining runs to complete sort
188     if (DEBUG) assert lo == hi;
189     ts.mergeForceCollapse();
190     if (DEBUG) assert ts.stackSize == 1;
191 }
192
193 /**
194  * Sorts the specified portion of the specified array using a binary
195  * insertion sort. This is the best method for sorting small
196  * numbers
197  * of elements. It requires O(n log n) compares, but O(n^2) data
198  * movement (worst case).
199  *
200  * If the initial part of the specified range is already sorted,
201  * this method can take advantage of it: the method assumes that the
202  * elements from index {@code lo}, inclusive, to {@code start},
203  * exclusive are already sorted.
204  *
205  * @param a the array in which a range is to be sorted
206  * @param lo the index of the first element in the range to be
207  * sorted
208  * @param hi the index after the last element in the range to be
209  * sorted
210  * @param start the index of the first element in the range that is
211  * not already known to be sorted {@code lo <= start <= hi}
212 */

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

208     @SuppressWarnings("fallthrough")
209     private static void binarySort(Object[] a, int lo, int hi, int
210         start) {
211         if (DEBUG) assert lo <= start && start <= hi;
212         if (start == lo)
213             start++;
214         for ( ; start < hi; start++) {
215             @SuppressWarnings("unchecked")
216             Comparable<Object> pivot = (Comparable) a[start];
217
218             // Set left (and right) to the index where a[start] (pivot)
219             // belongs
220             int left = lo;
221             int right = start;
222             if (DEBUG) assert left <= right;
223             /*
224             * Invariants:
225             *   pivot >= all in [lo, left].
226             *   pivot < all in [right, start].
227             */
228             while (left < right) {
229                 int mid = (left + right) >>> 1;
230                 if (pivot.compareTo(a[mid]) < 0)
231                     right = mid;
232                 else
233                     left = mid + 1;
234             }
235             if (DEBUG) assert left == right;
236
237             /*
238             * The invariants still hold: pivot >= all in [lo, left) and
239             * pivot < all in [left, start), so pivot belongs at left.
240             Note
241             * that if there are elements equal to pivot, left points to
242             * the
243             * first slot after them -- that's why this sort is stable.
244             * Slide elements over to make room to make room for pivot.
245             */
246             int n = start - left; // The number of elements to move
247             // Switch is just an optimization for arraycopy in default
248             // case
249             switch(n) {
250                 case 2: a[left + 2] = a[left + 1];
251                 case 1: a[left + 1] = a[left];
252                 break;
253                 default: System.arraycopy(a, left, a, left + 1, n);
254             }
255             a[left] = pivot;
256         }
257     }
258
259     /**
260      * Returns the length of the run beginning at the specified position
261      * in
262      * the specified array and reverses the run if it is descending
263      * (ensuring
264      * that the run will always be ascending when the method returns).
265      *
266      * A run is the longest ascending sequence with:
267      *

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

261      *      a[lo] <= a[lo + 1] <= a[lo + 2] <= ...
262      *
263      * or the longest descending sequence with:
264      *
265      *      a[lo] > a[lo + 1] > a[lo + 2] > ...
266      *
267      * For its intended use in a stable mergesort, the strictness of the
268      * definition of "descending" is needed so that the call can safely
269      * reverse a descending sequence without violating stability.
270      *
271      * @param a the array in which a run is to be counted and possibly
272      * reversed
273      * @param lo index of the first element in the run
274      * @param hi index after the last element that may be contained in
275      * the run.
276      *      It is required that @code{lo < hi}.
277      * @return the length of the run beginning at the specified
278      * position in
279      *      the specified array
280      */
281  @SuppressWarnings("unchecked")
282  private static int countRunAndMakeAscending(Object[] a, int lo, int
283  hi) {
284      if (DEBUG) assert lo < hi;
285      int runHi = lo + 1;
286      if (runHi == hi)
287          return 1;
288
289      // Find end of run, and reverse range if descending
290      if (((Comparable) a[runHi++]).compareTo(a[lo]) < 0) { // Descending
291          while(runHi < hi && ((Comparable)
292                  a[runHi]).compareTo(a[runHi - 1]) < 0)
293                  runHi++;
294          reverseRange(a, lo, runHi);
295      } else { // Ascending
296          while (runHi < hi && ((Comparable)
297                  a[runHi]).compareTo(a[runHi - 1]) >= 0)
298                  runHi++;
299      }
300
301      return runHi - lo;
302  }
303
304  /**
305   * Reverse the specified range of the specified array.
306   *
307   * @param a the array in which a range is to be reversed
308   * @param lo the index of the first element in the range to be
309   * reversed
310   * @param hi the index after the last element in the range to be
311   * reversed
312   */
313  private static void reverseRange(Object[] a, int lo, int hi) {
314      hi--;
315      while (lo < hi) {
316          Object t = a[lo];
317          a[lo++] = a[hi];
318          a[hi--] = t;
319      }
320  }

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\Com  
parableTimSort.java

---

```

312     }
313
314     /**
315      * Returns the minimum acceptable run length for an array of the
316      * specified
317      * length. Natural runs shorter than this will be extended with
318      * {@link #binarySort}.
319      *
320      * Roughly speaking, the computation is:
321      *
322      * If n < MIN_MERGE, return n (it's too small to bother with fancy
323      * stuff).
324      * Else if n is an exact power of 2, return MIN_MERGE/2.
325      * Else return an int k, MIN_MERGE/2 <= k <= MIN_MERGE, such that
326      * n/k
327      * is close to, but strictly less than, an exact power of 2.
328      *
329      * For the rationale, see listsort.txt.
330      *
331      * @param n the length of the array to be sorted
332      * @return the length of the minimum run to be merged
333      */
334     private static int minRunLength(int n) {
335         if (DEBUG) assert n >= 0;
336         int r = 0;          // Becomes 1 if any 1 bits are shifted off
337         while (n >= MIN_MERGE) {
338             r |= (n & 1);
339             n >>= 1;
340         }
341         return n + r;
342     }
343
344     /**
345      * Pushes the specified run onto the pending-run stack.
346      *
347      * @param runBase index of the first element in the run
348      * @param runLen the number of elements in the run
349      */
350     private void pushRun(int runBase, int runLen) {
351         this.runBase[stackSize] = runBase;
352         this.runLen[stackSize] = runLen;
353         stackSize++;
354     }
355
356     /**
357      * Examines the stack of runs waiting to be merged and merges
358      * adjacent runs
359      * until the stack invariants are reestablished:
360      *
361      * 1. runLen[i - 3] > runLen[i - 2] + runLen[i - 1]
362      * 2. runLen[i - 2] > runLen[i - 1]
363      *
364      * This method is called each time a new run is pushed onto the
365      * stack,
366      * so the invariants are guaranteed to hold for i < stackSize upon
367      * entry to the method.
368      */
369     private void mergeCollapse() {
370         while (stackSize > 1) {
371             int n = stackSize - 2;

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

367         if (n > 0 && runLen[n-1] <= runLen[n] + runLen[n+1]) {
368             if (runLen[n - 1] < runLen[n + 1])
369                 n--;
370             mergeAt(n);
371         } else if (runLen[n] <= runLen[n + 1]) {
372             mergeAt(n);
373         } else {
374             break; // Invariant is established
375         }
376     }
377 }
378
379 /**
380  * Merges all runs on the stack until only one remains.  This method
381  * is
382  * called once, to complete the sort.
383  */
384 private void mergeForceCollapse() {
385     while (stackSize > 1) {
386         int n = stackSize - 2;
387         if (n > 0 && runLen[n - 1] < runLen[n + 1])
388             n--;
389         mergeAt(n);
390     }
391 }
392 /**
393  * Merges the two runs at stack indices i and i+1.  Run i must be
394  * the penultimate or antepenultimate run on the stack.  In other
395  * words,
396  * i must be equal to stackSize-2 or stackSize-3.
397  *
398  * @param i stack index of the first of the two runs to merge
399  */
400 @SuppressWarnings("unchecked")
401 private void mergeAt(int i) {
402     if (DEBUG) assert stackSize >= 2;
403     if (DEBUG) assert i >= 0;
404     if (DEBUG) assert i == stackSize - 2 || i == stackSize - 3;
405
406     int base1 = runBase[i];
407     int len1 = runLen[i];
408     int base2 = runBase[i + 1];
409     int len2 = runLen[i + 1];
410     if (DEBUG) assert len1 > 0 && len2 > 0;
411     if (DEBUG) assert base1 + len1 == base2;
412
413     /*
414      * Record the length of the combined runs; if i is the 3rd-last
415      * run now, also slide over the last run (which isn't involved
416      * in this merge).  The current run (i+1) goes away in any case.
417      */
418     runLen[i] = len1 + len2;
419     if (i == stackSize - 3) {
420         runBase[i + 1] = runBase[i + 2];
421         runLen[i + 1] = runLen[i + 2];
422     }
423     stackSize--;
424 }

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

425         * Find where the first element of run2 goes in run1. Prior
426         * elements
427         * in run1 can be ignored (because they're already in place).
428         */
429     int k = gallopRight((Comparable<Object>) a[base2], a, base1,
430     len1, 0);
431     if (DEBUG) assert k >= 0;
432     base1 += k;
433     len1 -= k;
434     if (len1 == 0)
435         return;
436
437     /*
438     * Find where the last element of run1 goes in run2. Subsequent
439     * elements
440     * in run2 can be ignored (because they're already in place).
441     */
442     len2 = gallopLeft((Comparable<Object>) a[base1 + len1 - 1], a,
443         base2, len2, len2 - 1);
444     if (DEBUG) assert len2 >= 0;
445     if (len2 == 0)
446         return;
447
448     // Merge remaining runs, using tmp array with min(len1, len2)
449     // elements
450     if (len1 <= len2)
451         mergeLo(base1, len1, base2, len2);
452     else
453         mergeHi(base1, len1, base2, len2);
454 }
455
456 /**
457 * Locates the position at which to insert the specified key into
458 * the
459 * specified sorted range; if the range contains an element equal to
460 * key,
461 * returns the index of the leftmost equal element.
462 * @param key the key whose insertion point to search for
463 * @param a the array in which to search
464 * @param base the index of the first element in the range
465 * @param len the length of the range; must be > 0
466 * @param hint the index at which to begin the search, 0 <= hint <
467 * n.
468 *      The closer hint is to the result, the faster this method will
469 * run.
470 * @return the int k, 0 <= k <= n such that a[b + k - 1] < key <=
471 * a[b + k],
472 *      pretending that a[b - 1] is minus infinity and a[b + n] is
473 * infinity.
474 *      In other words, key belongs at index b + k; or in other words,
475 *      the first k elements of a should precede key, and the last n -
476 *      k
477 *      should follow it.
478 */
479 private static int gallopLeft(Comparable<Object> key, Object[] a,
480     int base, int len, int hint) {
481     if (DEBUG) assert len > 0 && hint >= 0 && hint < len;
482
483     int lastOfs = 0;

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

474         int ofs = 1;
475         if (key.compareTo(a[base + hint]) > 0) {
476             // Gallop right until a[base+hint+lastOfs] < key <=
477             // a[base+hint+ofs]
478             int maxOfs = len - hint;
479             while (ofs < maxOfs && key.compareTo(a[base + hint + ofs]) >
480                   0) {
481                 lastOfs = ofs;
482                 ofs = (ofs << 1) + 1;
483                 if (ofs <= 0) // int overflow
484                     ofs = maxOfs;
485             }
486             if (ofs > maxOfs)
487                 ofs = maxOfs;
488
489             // Make offsets relative to base
490             lastOfs += hint;
491             ofs += hint;
492         } else { // key <= a[base + hint]
493             // Gallop left until a[base+hint-ofs] < key <=
494             // a[base+hint-lastOfs]
495             final int maxOfs = hint + 1;
496             while (ofs < maxOfs && key.compareTo(a[base + hint - ofs]) <=
497                   0) {
498                 lastOfs = ofs;
499                 ofs = (ofs << 1) + 1;
500                 if (ofs <= 0) // int overflow
501                     ofs = maxOfs;
502             }
503             if (ofs > maxOfs)
504                 ofs = maxOfs;
505
506             // Make offsets relative to base
507             int tmp = lastOfs;
508             lastOfs = hint - ofs;
509             ofs = hint - tmp;
510         }
511         if (DEBUG) assert -1 <= lastOfs && lastOfs < ofs && ofs <= len;
512
513         /*
514          * Now a[base+lastOfs] < key <= a[base+ofs], so key belongs
515          * somewhere
516          * to the right of lastOfs but no farther right than ofs. Do a
517          * binary
518          * search, with invariant a[base + lastOfs - 1] < key <= a[base
519          * + ofs].
520          */
521         lastOfs++;
522         while (lastOfs < ofs) {
523             int m = lastOfs + ((ofs - lastOfs) >>> 1);
524
525             if (key.compareTo(a[base + m]) > 0)
526                 lastOfs = m + 1; // a[base + m] < key
527             else
528                 ofs = m; // key <= a[base + m]
529         }
530         if (DEBUG) assert lastOfs == ofs; // so a[base + ofs - 1] <
531         key <= a[base + ofs]
532         return ofs;
533     }

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

526
527     /**
528      * Like gallopLeft, except that if the range contains an element
529      * equal to
530      * key, gallopRight returns the index after the rightmost equal
531      * element.
532      *
533      * @param key the key whose insertion point to search for
534      * @param a the array in which to search
535      * @param base the index of the first element in the range
536      * @param len the length of the range; must be > 0
537      * @param hint the index at which to begin the search, 0 <= hint <
538      * n.
539      *      The closer hint is to the result, the faster this method will
540      * run.
541      * @return the int k, 0 <= k <= n such that a[b + k - 1] <= key <
542      * a[b + k]
543      */
544  private static int gallopRight(Comparable<Object> key, Object[] a,
545      int base, int len, int hint) {
546      if (DEBUG) assert len > 0 && hint >= 0 && hint < len;
547
548      int ofs = 1;
549      int lastOfs = 0;
550      if (key.compareTo(a[base + hint]) < 0) {
551          // Gallop left until a[b+hint - ofs] <= key < a[b+hint -
552          lastOfs]
553          int maxOfs = hint + 1;
554          while (ofs < maxOfs && key.compareTo(a[base + hint - ofs]) <
555          0) {
556              lastOfs = ofs;
557              ofs = (ofs << 1) + 1;
558              if (ofs <= 0) // int overflow
559                  ofs = maxOfs;
560          }
561          if (ofs > maxOfs)
562              ofs = maxOfs;
563
564          // Make offsets relative to b
565          int tmp = lastOfs;
566          lastOfs = hint - ofs;
567          ofs = hint - tmp;
568      } else { // a[b + hint] <= key
569          // Gallop right until a[b+hint + lastOfs] <= key < a[b+hint
570          + ofs]
571          int maxOfs = len - hint;
572          while (ofs < maxOfs && key.compareTo(a[base + hint + ofs]) >=
573          0) {
574              lastOfs = ofs;
575              ofs = (ofs << 1) + 1;
576              if (ofs <= 0) // int overflow
577                  ofs = maxOfs;
578          }
579          if (ofs > maxOfs)
580              ofs = maxOfs;
581
582          // Make offsets relative to b
583          lastOfs += hint;
584          ofs += hint;
585      }
586  }

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

577         if (DEBUG) assert -1 <= lastOfs && lastOfs < ofs && ofs <= len;
578
579         /*
580          * Now a[b + lastOfs] <= key < a[b + ofs], so key belongs
581          * somewhere to
582          * the right of lastOfs but no farther right than ofs.  Do a
583          * binary
584          * search, with invariant a[b + lastOfs - 1] <= key < a[b +
585          * ofs].
586          */
587         lastOfs++;
588         while (lastOfs < ofs) {
589             int m = lastOfs + ((ofs - lastOfs) >>> 1);
590
591             if (key.compareTo(a[base + m]) < 0)
592                 ofs = m;           // key < a[b + m]
593             else
594                 lastOfs = m + 1; // a[b + m] <= key
595         }
596
597         if (DEBUG) assert lastOfs == ofs; // so a[b + ofs - 1] <= key
598         < a[b + ofs]
599         return ofs;
600     }
601
602     /**
603      * Merges two adjacent runs in place, in a stable fashion.  The
604      * first
605      * element of the first run must be greater than the first element
606      * of the
607      * second run (a[base1] > a[base2]), and the last element of the
608      * first run
609      * (a[base1 + len1-1]) must be greater than all elements of the
610      * second run.
611      *
612      * For performance, this method should be called only when len1 <=
613      * len2;
614      * its twin, mergeHi should be called if len1 >= len2.  (Either
615      * method
616      * may be called if len1 == len2.)
617      *
618      * @param base1 index of first element in first run to be merged
619      * @param len1 length of first run to be merged (must be > 0)
620      * @param base2 index of first element in second run to be merged
621      * (must be aBase + aLen)
622      * @param len2 length of second run to be merged (must be > 0)
623      */
624     @SuppressWarnings("unchecked")
625     private void mergeLo(int base1, int len1, int base2, int len2) {
626         if (DEBUG) assert len1 > 0 && len2 > 0 && base1 + len1 == base2;
627
628         // Copy first run into temp array
629         Object[] a = this.a; // For performance
630         Object[] tmp = ensureCapacity(len1);
631         System.arraycopy(a, base1, tmp, 0, len1);
632
633         int cursor1 = 0;      // Indexes into tmp array
634         int cursor2 = base2; // Indexes into a
635         int dest = base1;    // Indexes into a
636
637         // Move first element of second run and deal with degenerate

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

        cases
627      a[dest++] = a[cursor2++];
628      if (--len2 == 0) {
629          System.arraycopy(tmp, cursor1, a, dest, len1);
630          return;
631      }
632      if (len1 == 1) {
633          System.arraycopy(a, cursor2, a, dest, len2);
634          a[dest + len2] = tmp[cursor1]; // Last elt of run 1 to end
635          of merge
636          return;
637      }
638      int minGalloping = this.minGalloping; // Use local variable for
639      performance
640      outer:
641      while (true) {
642          int count1 = 0; // Number of times in a row that first run
643          won
644          int count2 = 0; // Number of times in a row that second run
645          won
646          /*
647          * Do the straightforward thing until (if ever) one run
648          starts
649          * winning consistently.
650          */
651          do {
652              if (DEBUG) assert len1 > 1 && len2 > 0;
653              if (((Comparable) a[cursor2]).compareTo(tmp[cursor1]) <
654                  0) {
655                  a[dest++] = a[cursor2++];
656                  count2++;
657                  count1 = 0;
658                  if (--len2 == 0)
659                      break outer;
660              } else {
661                  a[dest++] = tmp[cursor1++];
662                  count1++;
663                  count2 = 0;
664                  if (--len1 == 1)
665                      break outer;
666              }
667          } while ((count1 | count2) < minGalloping);
668          /*
669          * One run is winning so consistently that galloping may be
670          a
671          * huge win. So try that, and continue galloping until (if
672          ever)
673          * neither run appears to be winning consistently anymore.
674          */
675          do {
676              if (DEBUG) assert len1 > 1 && len2 > 0;
677              count1 = gallopRight((Comparable) a[cursor2], tmp,
678                  cursor1, len1, 0);
679              if (count1 != 0) {
680                  System.arraycopy(tmp, cursor1, a, dest, count1);
681                  dest += count1;
682                  cursor1 += count1;
683              }
684          } while ((count1 | count2) < minGalloping);
685      }
686  }

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

677             len1 -= count1;
678             if (len1 <= 1) // len1 == 1 || len1 == 0
679                 break outer;
680             }
681             a[dest++] = a[cursor2++];
682             if (--len2 == 0)
683                 break outer;
684
685             count2 = gallopLeft((Comparable) tmp[cursor1], a,
686             cursor2, len2, 0);
687             if (count2 != 0) {
688                 System.arraycopy(a, cursor2, a, dest, count2);
689                 dest += count2;
690                 cursor2 += count2;
691                 len2 -= count2;
692                 if (len2 == 0)
693                     break outer;
694             }
695             a[dest++] = tmp[cursor1++];
696             if (--len1 == 1)
697                 break outer;
698             minGallop--;
699             } while (count1 >= MIN_GALLOP | count2 >= MIN_GALLOP);
700             if (minGallop < 0)
701                 minGallop = 0;
702             minGallop += 2; // Penalize for leaving gallop mode
703             } // End of "outer" loop
704             this.minGallop = minGallop < 1 ? 1 : minGallop; // Write back
705             to field
706
707             if (len1 == 1) {
708                 if (DEBUG) assert len2 > 0;
709                 System.arraycopy(a, cursor2, a, dest, len2);
710                 a[dest + len2] = tmp[cursor1]; // Last elt of run 1 to end
711                 of merge
712             } else if (len1 == 0) {
713                 throw new IllegalArgumentException(
714                     "Comparison method violates its general contract!");
715             } else {
716                 if (DEBUG) assert len2 == 0;
717                 if (DEBUG) assert len1 > 1;
718                 System.arraycopy(tmp, cursor1, a, dest, len1);
719             }
720
721             /**
722             * Like mergeLo, except that this method should be called only if
723             * len1 >= len2; mergeLo should be called if len1 <= len2. (Either
724             * method
725             * may be called if len1 == len2.)
726             *
727             * @param base1 index of first element in first run to be merged
728             * @param len1 length of first run to be merged (must be > 0)
729             * @param base2 index of first element in second run to be merged
730             *         (must be aBase + aLen)
731             * @param len2 length of second run to be merged (must be > 0)
732             */
733             @SuppressWarnings("unchecked")
734             private void mergeHi(int base1, int len1, int base2, int len2) {
735                 if (DEBUG) assert len1 > 0 && len2 > 0 && base1 + len1 == base2;

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

733
734     // Copy second run into temp array
735     Object[] a = this.a; // For performance
736     Object[] tmp = ensureCapacity(len2);
737     System.arraycopy(a, base2, tmp, 0, len2);
738
739     int cursor1 = base1 + len1 - 1; // Indexes into a
740     int cursor2 = len2 - 1; // Indexes into tmp array
741     int dest = base2 + len2 - 1; // Indexes into a
742
743     // Move last element of first run and deal with degenerate cases
744     a[dest--] = a[cursor1--];
745     if (--len1 == 0) {
746         System.arraycopy(tmp, 0, a, dest - (len2 - 1), len2);
747         return;
748     }
749     if (len2 == 1) {
750         dest -= len1;
751         cursor1 -= len1;
752         System.arraycopy(a, cursor1 + 1, a, dest + 1, len1);
753         a[dest] = tmp[cursor2];
754         return;
755     }
756
757     int minGalloping = this.minGalloping; // Use local variable for
758     performance
759     outer:
760     while (true) {
761         int count1 = 0; // Number of times in a row that first run
762         won
763         int count2 = 0; // Number of times in a row that second run
764         won
765
766         /*
767          * Do the straightforward thing until (if ever) one run
768          * appears to win consistently.
769         */
770         do {
771             if (DEBUG) assert len1 > 0 && len2 > 1;
772             if (((Comparable) tmp[cursor2]).compareTo(a[cursor1]) <
773                 0) {
774                 a[dest--] = a[cursor1--];
775                 count1++;
776                 count2 = 0;
777                 if (--len1 == 0)
778                     break outer;
779             } else {
780                 a[dest--] = tmp[cursor2--];
781                 count2++;
782                 count1 = 0;
783                 if (--len2 == 1)
784                     break outer;
785             }
786         } while ((count1 | count2) < minGalloping);
787
788         /*
789          * One run is winning so consistently that galloping may be
790          * a
791          * huge win. So try that, and continue galloping until (if
792          * ever)
793     }

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

787             * neither run appears to be winning consistently anymore.
788             */
789     do {
790         if (DEBUG) assert len1 > 0 && len2 > 1;
791         count1 = len1 - gallopRight((Comparable) tmp[cursor2],
792             a, base1, len1, len1 - 1);
793         if (count1 != 0) {
794             dest -= count1;
795             cursor1 -= count1;
796             len1 -= count1;
797             System.arraycopy(a, cursor1 + 1, a, dest + 1,
798                             count1);
799             if (len1 == 0)
800                 break outer;
801         }
802         a[dest--] = tmp[cursor2--];
803         if (--len2 == 1)
804             break outer;
805         count2 = len2 - gallopLeft((Comparable) a[cursor1], tmp,
806             0, len2, len2 - 1);
807         if (count2 != 0) {
808             dest -= count2;
809             cursor2 -= count2;
810             len2 -= count2;
811             System.arraycopy(tmp, cursor2 + 1, a, dest + 1,
812                             count2);
813             if (len2 <= 1)
814                 break outer; // len2 == 1 || len2 == 0
815         }
816         a[dest--] = a[cursor1--];
817         if (--len1 == 0)
818             break outer;
819         minGallop--;
820     } while (count1 >= MIN_GALLOP | count2 >= MIN_GALLOP);
821     if (minGallop < 0)
822         minGallop = 0;
823     minGallop += 2; // Penalize for leaving gallop mode
824     } // End of "outer" loop
825     this.minGallop = minGallop < 1 ? 1 : minGallop; // Write back
826     to field
827
828     if (len2 == 1) {
829         if (DEBUG) assert len1 > 0;
830         dest -= len1;
831         cursor1 -= len1;
832         System.arraycopy(a, cursor1 + 1, a, dest + 1, len1);
833         a[dest] = tmp[cursor2]; // Move first elt of run2 to front
834         of merge
835     } else if (len2 == 0) {
836         throw new IllegalArgumentException(
837             "Comparison method violates its general contract!");
838     } else {
839         if (DEBUG) assert len1 == 0;
840         if (DEBUG) assert len2 > 0;
841         System.arraycopy(tmp, 0, a, dest - (len2 - 1), len2);
842     }
843 }
844 /**

```

G:\eclair21 - GOOGLE-00-0000525\dalvik\libcore\luni\src\main\java\java\util\ComparableTimSort.java

---

```

841     * Ensures that the external array tmp has at least the specified
842     * number of elements, increasing its size if necessary. The size
843     * increases exponentially to ensure amortized linear time
844     * complexity.
845     *
846     * @param minCapacity the minimum required capacity of the tmp array
847     * @return tmp, whether or not it grew
848     */
849     private Object[] ensureCapacity(int minCapacity) {
850         if (tmp.length < minCapacity) {
851             // Compute smallest power of 2 > minCapacity
852             int newSize = minCapacity;
853             newSize |= newSize >> 1;
854             newSize |= newSize >> 2;
855             newSize |= newSize >> 4;
856             newSize |= newSize >> 8;
857             newSize |= newSize >> 16;
858             newSize++;
859
860             if (newSize < 0) // Not bloody likely!
861                 newSize = minCapacity;
862             else
863                 newSize = Math.min(newSize, a.length >>> 1);
864
865             @SuppressWarnings({ "unchecked", "UnnecessaryLocalVariable" })
866             Object[] newArray = new Object[newSize];
867             tmp = newArray;
868         }
869         return tmp;
870     }
871
872     /**
873     * Checks that fromIndex and toIndex are in range, and throws an
874     * appropriate exception if they aren't.
875     *
876     * @param arrayLen the length of the array
877     * @param fromIndex the index of the first element of the range
878     * @param toIndex the index after the last element of the range
879     * @throws IllegalArgumentException if fromIndex > toIndex
880     * @throws ArrayIndexOutOfBoundsException if fromIndex < 0
881     *         or toIndex > arrayLen
882     */
883     private static void rangeCheck(int arrayLen, int fromIndex, int
884     toIndex) {
885         if (fromIndex > toIndex)
886             throw new IllegalArgumentException("fromIndex(" + fromIndex
887             +
888             ") > toIndex(" + toIndex + ")");
889         if (fromIndex < 0)
890             throw new ArrayIndexOutOfBoundsException(fromIndex);
891         if (toIndex > arrayLen)
892             throw new ArrayIndexOutOfBoundsException(toIndex);
893     }
894 }
```